

1. [Lab 0: DSP Hardware Introduction](#)
2. [Lab 1: Prelab](#)
3. [Lab 1: Lab](#)
4. [Lab 2: Prelab part I \(multirate theory\)](#)
5. [Lab 2 : Prelab part II \(upsampling and downsampling\)](#)
6. [Lab 2: Multirate Filtering](#)
7. [Lab 3: Prelab \(Part 1\)](#)
8. [Lab 3: Prelab \(Part 2\)](#)
9. [Lab 3: Lab](#)
10. [Lab 4 Prelab 4](#)
11. [Lab 4: Lab](#)
12. [Lab 5: Prelab 5](#)
13. [Lab 5: Histogram equalization](#)

Lab 0: DSP Hardware Introduction

This exercise introduces the DSP hardware and associated software used in the course. By the end of this module, you should be comfortable with the basics of testing a simple real-time DSP system with Code Composer Studio, the TI DSP c55x debugging environment. First you will connect the laboratory equipment and test a real-time DSP system with provided code to implement an eight-tap (eight coefficient) finite impulse response (FIR) filter. With a working system available, you will then begin to explore the debugging software used for downloading, modifying, and testing your code. Finally, you will create a filter in MATLAB and use test vectors to verify the DSP's output.

Introduction

This exercise introduces the work-flow environment for developing real-time signal-processing systems, which consists of three major components:

1. hardware I/O tools (i.e., a function generator and oscilloscope), which are valuable for testing the functionality of a real-time DSP system,
2. a software debugging environment, such as Code Composer Studio (CCS), which is used to write, download, execute, and debug DSP code,
3. a high-level development environment, such as MATLAB, which is used to verify computational correctness, and explore and debug conceptual issues.

The DSP task in this exercise will be to implement a pre-written eight-tap (eight coefficient) **finite impulse response (FIR)** filter, verifying correctness using the hardware I/O tools, along with MATLAB test vectors.

Lab Equipment

This exercise assumes you have access to a laboratory station equipped with a Texas Instruments TMS320C5510A-200 digital signal processor chip mounted on a Spectrum Digital TMS320VC5510 evaluation board. The DUAL3006, a daughtercard produced by Educational DSP, is mounted on the external peripheral interface of the board to enable four-input/four-

output capability. The evaluation module should be connected to a PC running Windows and will be controlled using CCS v5.0. We will be using a 48kHz sample rate.

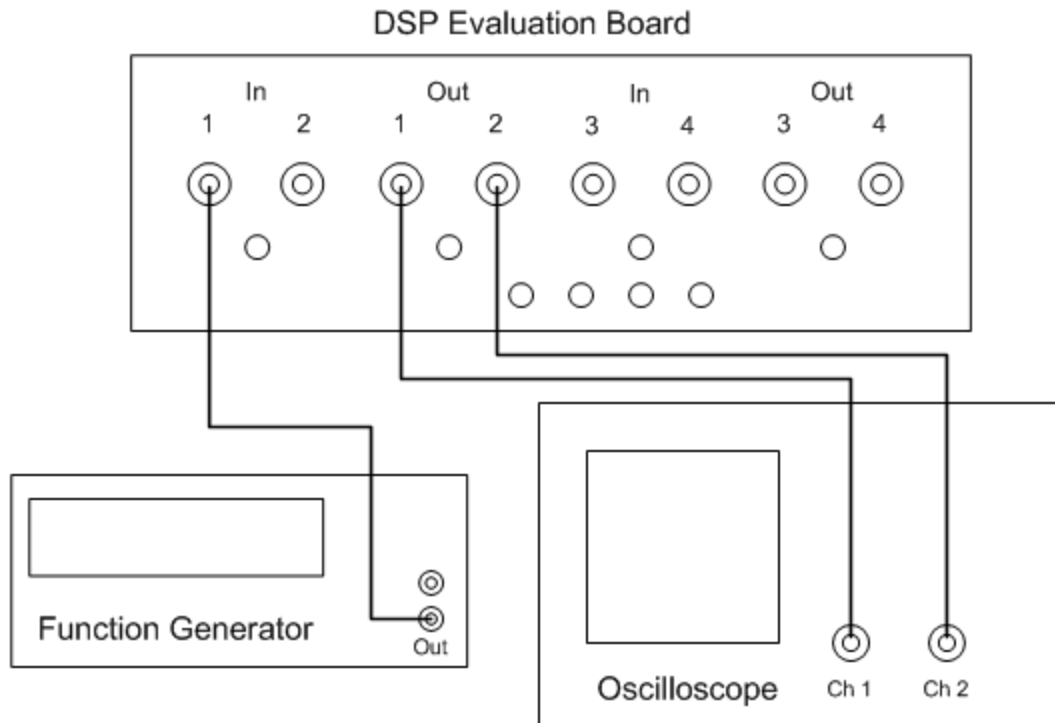
Note: If you are not using Code Composer Studio v5.0, the instructions on this page do not apply. Please see the [missing_resource: <http://cnx.org/content/m13811/latest/?collection=col10397>]

In addition to the DSP board and PC, each laboratory station should also be equipped with a function generator to provide test signals and an oscilloscope to display the processed waveforms.

Step 1: Connect cables

Use the provided BNC cables to connect the output of the function generator to the input of the DSP evaluation board, and the DSP outputs to the oscilloscope, as shown in [\[link\]](#).

Example Hardware Setup



Note: The actual channel ports may differ than what is shown in the illustration. Read the labels on the evaluation board!

With this configuration, you will have only one signal going into the DSP board and two signals coming out. The output on channel 1 is the filtered input signal, and the output on channel 2 is the unfiltered input signal. This allows you to view the raw input and filtered output simultaneously on the oscilloscope. Turn on the function generator and the oscilloscope.

Step 2: Log in

Use your netID and password to log into the PC at your laboratory station.

When you log in, verify that the **U:** and **V:** networked drives are mapped to the computer. The **U:** drive is your personal work directory, and **V:** is a

read-only course drive.

Although you may want to work exclusively in one partner's network account, you should be sure that both partners have copies of the lab assignment code.

The Development Environment

The evaluation board is controlled by the PC through the JTAG interface in CCS. This development environment allows the user to download, run, and debug code assembled on the PC. Work through the steps below to familiarize yourself with the debugging environment and real-time system using the provided FIR filter code (Steps 3, 4 and 5), then verify the filter's frequency response with the subsequent MATLAB exercises (Steps 6 and 7).

Step 3: Assemble filter code

Setup Code Composer

By default, a shortcut to CCS is available by going to **Start > All Programs > Texas Instruments > Code Composer Studio v5**. When CCS starts for the first time, Workspace Launcher will start because it will need to set up your workspace.

Create or make sure you have the following directory:
U:\workspace\ECE420. In Workspace Launcher, hit **Browse...**, navigate to this folder, and make sure to check "Use this as the default and do not ask again".

Note: In the future, verify that you are in the correct workspace by going to **File > Switch Workspace...**

Note: Make sure the workspace path does not start with "`\ad.uillinois.edu...`"

Import Project

In CCS, go to **Project > Import Existing CCS Eclipse Project**:

1. Browse to `V:\ece420\55x\ccs5\filter`
2. Check "Copy projects into workspace"

Build Project

Once the project is copied into your workspace, we can proceed to build it:

- In Project Explorer, make sure that "filter" is highlighted.
- Select **Project > Build Project**.

In a successful build, there will be zero errors and maybe a few warnings and remarks. The output file will be placed in a Debug folder within the project's directory. In this example, the executable binary code will be located at `.\Debug\filter.out`.

Step 4: Verify filter execution

Connect to the DSP

1. Select **View > Target Configurations**
2. In the panel that comes up, expand **Projects > filter**
3. Right-click on `dsk5510.ccxml` and select "Launch Selected Configuration"

Once CCS connects to the DSP, the debugger view will launch. Select **Run > Connect Target**

Load and Run Program

Now, load your assembled filter file onto the DSP:

- Select **Run > Load > Load Program**.
- Select "Browse project" and choose the binary file **filter.out**.
- Execute the code by selecting **Run > Resume**.

The program you are running accepts input from input channel 1 and sends output waveforms to output channels 1 and 2 (the filtered signal and raw input, respectively).

Exercise:

Problem:

Note that the "raw input" on output channel 2 may differ from the actual input on input channel 1. Why?

Solution:

Because of distortions introduced in converting the analog input to a digital signal and then back to an analog signal.

Exercise:

Problem:

What differences do you expect to see between the signals at input channel 1 and at output channel 2?

Solution:

Hint: The A/D and D/A converters on the six-channel surround board operate at a sample rate of 48 kHz and have an **anti-aliasing filter** and an **anti-imaging filter**, respectively, that in the ideal case would eliminate frequency content above 24 kHz.

Exercise:

Problem:

How would output channel 2 be different if the input had a DC offset?

Solution:

the converters on the board are **AC coupled** and cannot pass DC signals.

Configure Function Generator and Oscilloscope

Set the amplitude on the function generator to 1.0 V peak-to-peak and the pulse shape to sinusoidal. Adjust the function generator so that it expects a high impedance load. The sequence of button presses to accomplish this on the function generator in the lab is **Shift -> Enter -> Right -> Right -> Right -> Down -> Down -> Right -> Enter**.

Make sure the oscilloscope is set to 1M impedance. This can be accomplished by pressing channel 1 or 2 and then selecting 1M Ohm from the Imped menu.

Observe the frequency response of the filter by sweeping the input signal through the relevant frequency range. What is the relevant frequency range for a DSP system with a sample rate of 48 kHz?

Characterize Filter Response

Based on the frequency response you observe, characterize the filter in terms of its type (e.g., low-pass, high-pass, band-pass) and its -6 dB (half-amplitude) cutoff frequency (or frequencies). It may help to set the trigger on channel 2 of the oscilloscope since the signal on channel 1 may go to zero.

Step 5: Re-assemble and re-run with new filter

Once you have determined the type of filter the DSP is implementing, you are ready to repeat the process with a different filter by including different coefficients during the assembly process. There is a second set of filter coefficients already in your project folder. In Windows Explorer, navigate to **U:\workspace\ece420\filter** and do the following:

- Rename `coef.asm` to `coef1.asm`
- Rename `coef2.asm` to `coef.asm`

Repeat the assembly and testing process with the new filter by repeating steps required to build ([Step 3](#)) and execute ([Step 4](#)) the code. (You will need to rebuild filtercode.asm manually by right-clicking this file and selecting "Build Selected File". Afterwards, rebuild the project. There is a bug in CCSv5's Makefile generation that fails to make filtercode.asm depend on changes to coef.asm")

Just as you did in [Step 4](#), determine the type of filter you are running and the filter's -6 dB point by testing the system at various frequencies.

Step 6: Check filter response in MATLAB

In this step, you will use MATLAB to verify the frequency response of your filter by copying the coefficients from the DSP to MATLAB and displaying the magnitude of the frequency response using the MATLAB command `freqz`.

View Coefficients in DSP Memory

The FIR filter coefficients included in the file `coef.asm` are stored in memory on the DSP. To view the contents of the DSP memory, first suspend any running program by going to `Run > Suspend` and then select `View > Memory Browser`.

In the panel that comes up, there is a text box for you to type in the name of the variable that you are interested in viewing. This variable name is actually a mnemonic for a memory address. In the case of our coefficients, the mnemonic `coef1` is used to point to the starting address of our coefficients. The memory content can be displayed in many different formats. In the drop-down box, choose `16-Bit Signed Int`.

Note: Make sure you understand where the `coef1` label comes from. [Hint:] Select `View > C/C++ Projects` and double click on `filtercode.asm` to view the source code.

In this example, the filter coefficients are placed in memory in decreasing order; that is, the last coefficient, $h[7]$, is at location `coef1` and the first coefficient, $h[0]$, is stored at `coef1+7`.

Now that you can find the coefficients in memory, you are ready to use the MATLAB command `freqz` to view the filter's response. You must create a vector in MATLAB with the filter coefficients to use the `freqz` command. For example, if you want to view the response of the three-tap filter with coefficients -10, 20, -10 you can use the following commands in MATLAB:

- `h = [-10, 20, -10];`
- `freqz(h)`

Note that you will have to enter eight values, the contents of memory locations `coef1` through `coef1+7`, into the coefficient vector, `h`.

Note: You must divide the coefficients by 32768. Where does this scaling factor come from?

How does the MATLAB response compare with your experimental results? What might account for any differences?

Step 7: Create new filter in MATLAB and verify

MATLAB scripts will be made available to you to aid in code development. For example, one of these scripts allows you to save filter coefficients

created in MATLAB in a form that can be included as part of the assembly process without having to type them in by hand (a very useful tool for long filters). These scripts may already be installed on your computer; otherwise, download the files from the links as they are introduced.

First, have MATLAB generate a "random" eight-tap filter by typing `h = gen_filt;` at a MATLAB prompt. Then save this vector of filter coefficients by typing `save_coef('coef.asm', flip1r(h));` Make sure you save the file in your own directory. (The scripts that perform these functions are available as `gen_filt.m` and `save_coef.m` and can be found at [V:/ece420/55x/m_files](#))

The `save_coef` MATLAB script will save the coefficients of the vector `h` into the named file, which in this case is `coef.asm`. Note that the coefficient vector is "flipped" prior to being saved; this is to make the coefficients in `h` fill DSP memory-locations `coef1` through `coef1+7` in reverse order, as before.

You may now re-assemble and re-run your new filter code as you did in [Step 5](#).

Notice when you load your new filter that the contents of memory locations `coef1` through `coef1+7` update accordingly.

Step 8: Modify filter coefficients in memory

Not only can you view the contents of memory on the DSP using the debugger, you can change the contents at any memory location simply by double-clicking on the location and making the desired change in the pop-up window.

Note: The DSP must be in a halted state in order to overwrite the memory.

Change the contents of memory locations `coef1` through `coef1+7` such that the coefficients implement a scale and delay filter with impulse response:

Equation:

$$h[n] = 8192\delta[n - 4]$$

Note that the DSP interprets the integer value of 8192 as a fractional number by dividing the integer by 32,768 (the largest integer possible in a 16-bit two's complement register). The result is an output that is delayed by four samples and scaled by a factor of $\frac{1}{4}$. More information on the DSP's interpretation of numbers appears in [Two's Complement and Fractional Arithmetic for 16-bit Processors](#).

Note: A clear and complete understanding of how the DSP interprets numbers is absolutely necessary to effectively write programs for the DSP. Save yourself time later by learning this material before attempting Lab 1!

After you have made the changes to all eight coefficients, run your new filter and use the oscilloscope to measure the delay between the raw (input) and filtered (delayed) waveforms.

Note: Take advantage of the "Quick Measure" feature on the oscilloscope!

What happens to the output if you change either the scaling factor or the delay value? How many seconds long is a single-sample delay? Six-sample delay?

Step 9: Test-vector simulation

As a final exercise, you will find the output of the DSP for an input specified by a test vector. Then you will compare that output with the output of a MATLAB simulation of the same filter processing the same input; if the DSP implementation is correct, the two outputs should be almost identical. To do this, you will generate a waveform in MATLAB and save it as a test vector. You will then run your DSP filter using the test vector as input and import the results back into MATLAB for comparison with a MATLAB simulation of the filter.

The first step in using test vectors is to generate an appropriate input signal. One way to do this is to use the MATLAB function to generate a sinusoid that sweeps across a range of frequencies. The MATLAB function `save_test_vector` (available as `save_test_vector.m` under `m_files`) can then save the sinusoidal sweep to a file you will later include in the DSP code.

Generate a sinusoidal sweep using `sweep.m` and save it to a DSP test-vector file using the following MATLAB commands:

- `t=sweep(0.1*pi,0.9*pi,0.25,500); % Generate a frequency sweep`
- `save_test_vector('testvect.asm',t); % Save the test vector`

Next, use the MATLAB `conv` command to generate a simulated response by filtering the sweep with the filter h you generated using `gen_filt` above. Note that this operation will yield a vector of length 507 (which is $n + m - 1$, where n is the length of the filter and m is the length of the input). You should keep only the first 500 elements of the resulting vector.

- `out=conv(h,t); % Filter t with FIR filter h`
- `out=out(1:500); % Keep first 500 elements of out`

The `main.c` file needs to be told to take input from memory on the DSP. Fortunately, the changes have already been made in the files. The test vector is stored in a block of memory on the DSP just like other variables. The memory block that holds the test vector is large enough to hold a vector up to 4,000 elements long. The test vector stores data for all four channels of input and from four channels of output.

To run your program with test vectors, you will need to modify `main.c` as well as `filtercode.asm`. Both are simply text files and can be edited using the editor of your preference, including WordPad, Emacs, and VI. (The changes have already been made, but please visually verify the changes are there.) Within `main.c`, uncomment the `#define FILE_INPUT` line so that your program will rewrite input from the A/D with the test vector you specified and then save the output into a block of memory.

In `filtercode.asm`, uncomment the `.copy "testvect.asm"` line. Make sure this Matlab generated file is in the same directory as `filtercode.asm`.

Note: In TI assembly, the semi-colon `;` signifies a comment.

These changes will copy in the test vector. After modifying your code, assemble it, then load and run the file using Code Composer as before. After a few seconds, halt the DSP (using the `Suspend` command under the `Run` menu). How many seconds do you think it should take?

Saving DSP Memory to File

Next, we will save the test output file and load it back into MATLAB. We are interested in the first 500 output samples, starting at address `tv_outbuf` in `Data` memory. There are four output channels and the memory is interleaved in time. Therefore, we will have to collect 2000 (4 channels time 500 samples) memory elements.

- Select **View > Memory Browser**
- Click on the "Save" icon, a green square with an angled arrow (top left in the Memory panel)
- Hit Browse and make sure you are in your U: workspace
- Name the file **output.dat** and save filetype as TI data format
- On the next screen, use the following options:
 - format: **16-Bit Hex - TI Style**
 - start address: **tv_outbuf**
 - memory page: **data**
 - length: **2000**

Last, use the **read_vector** (available as **read_vector.m**) function to read the saved result into MATLAB. Do this using the following MATLAB command:

- **[ch1,ch2,ch3,ch4] = read_vector('output.dat');**

Now, the MATLAB vector **ch1** corresponds to the filtered version of the test signal you generated. The MATLAB vector **ch2** should be nearly identical to the test vector you generated, as it was passed from the DSP system's input to its output unchanged.

Note: Because of quantization error introduced in saving the test vector for the 16-bit memory of the DSP, the vector **ch2** will not be identical to the MATLAB generated test vector.

After loading the output of the filter into MATLAB, compare the expected output (calculated as **out** above) and the output of the filter (in **ch1** from above). This can be done graphically by simply plotting the two curves on the same axes; for example:

- **plot(out,'r');** % Plot the expected curve in red
- **hold on** % Plot the next plot on top of this one

- `plot(ch1, 'g');` % Plot the expected curve in green
- `hold off`

You should also ensure that the difference between the two outputs is near zero. This can be done by plotting the difference between the two vectors:

- `plot(out(1:length(ch1))-ch1);` % Plot error signal

You will observe that the two sequences are not exactly the same; this is due to the fact that the DSP computes its response to 16 bits precision, while MATLAB uses 64-bit floating point numbers for its arithmetic. Blocks of output samples may also be missing from the test vector output due to a bug in the test vector core. Nonetheless, the test vector environment allows one to run repeatable experiments using the same known test input for debugging.

Step 10: Closing Down

Before exiting Code Composer, make sure to disconnect properly from the DSP:

- Halt any program running on the DSP (`Run > Suspend`)
- Disconnect from the DSP (`Run > Connect` will toggle between connecting and disconnecting)

Lab 1: Prelab

You will work through a section of TI TMS320C55x assembly code by hand. The instructions include multiplication of fractional numbers in two's complement representation.

Assembly Exercise

Analyze the following lines of code. Refer to [Two's Complement and Fractional Arithmetic for 16-bit Processors](#), [Addressing Modes for TI TMS320C55x](#), and the [Mnemonic Instruction Set](#) manual for help.

```
1  FIR_len .set      3
2
3  ; Assume:
4  ;   BK03 = FIR_len
5  ;   firStateIndex is stored at memory
location 1008h
6  ;   AR2 = 1000h
7  ;   AR3 = 1004h
8  ;   FRCT = 1
9
10      BSET      AR3LC                ; sets
circular addressing for AR3
11      mov      mmap(AR3), BSA23
12      mov      #firStateIndex, AR4
13      mov      *AR4, AR3
14      mov      LO(AC0), *AR3+
15      mov      #0, AC0
16      rpt      #(FIR_len-1)
17      macm      *AR2+, *AR3+, AC0
```

Anything following a ";" is considered a comment. In this case, the comments indicate the contents of the auxiliary registers, the **BK03** register, and the address registers before the execution of the first instruction, **mov**.

The line `FIR_len .set 3` defines the name `FIR_len` as equal to 3. The `BK03` register contains the length of the circular buffer we want to use for auxiliary register 0 through 3. The `BSET AR3LC` modifies the increment operator `+` so that it behaves as a circular buffer. This means circular addressing will be used for AR3. Refer to Section 6.11 of the [CPU Reference Guide](#) for help on circular addressing.

Note that any number followed by an "h" or preceded with a `0x` represents a **hexadecimal** value.

Example:

1000h and 0x1000 both refer to the decimal number 4096.

Assume that the data memory is initialized as follows starting at location `1000h`.

Memory location	Value
1000h	1000h
1001h	0000h
1002h	4000h
1004h	1000h
1005h	1000h

Memory location	Value
1006h	4000h
1007h	1000h
1008h	0000h

Data Memory Assignment (before execution)Data Memory Assignment (before execution)

After familiarizing yourself with the **mov**, **rpt**, and **macm** instructions, step through each line of code and record the values of the accumulator **AC0** and auxiliary registers **AR2** and **AR3** in the spaces provided in [\[link\]](#). Additionally, record the value of the memory contents after all three instructions have been "executed" in the blank data memory table in [\[link\]](#).

AC0	AR2	AR3	
00 0000 8000h	1000h	1004h	at start of code
			after mov instruction line 11

AC0	AR2	AR3	
			after mov instruction line 12
			after mov instruction line 13
			after mov instruction line 14
			after mov instruction line 15
			after rpt instruction line 16
			after first macm instruction

AC0	AR2	AR3	
			after second macm instruction
			after third macm instruction

Execution Results

When working through the exercise, take into account that the accumulator **AC0** is a 40-bit register, and that the multiplier is in the **fractional arithmetic mode**. In this mode, integers on the DSP are interpreted as fractions, and the multiplier will treat them accordingly. This is done by shifting the result of the integer multiplier in the ALU left one bit. (All the arithmetic is fractional in these examples.) Multiplies performed by the ALU (via the **macm** instruction) produce a result that is twice what you would expect if you just multiplied the two integers together. DSP numerical representation and arithmetic are described further in [Two's Complement and Fractional Arithmetic for 16-bit Processors](#).

Memory location	Value
1000h	
1001h	

Memory location	Value
1002h	
1004h	
1005h	
1006h	
1007h	
1008h	

Data Memory Assignment (after execution)Data Memory Assignment
(after execution)

Lab 1: Lab

You will implement band-pass finite impulse-response (FIR) filters with time-domain processing.

Introduction

In this exercise, you will program in the DSP's assembly language and C to create FIR filters. Begin by studying the assembly code for the basic FIR filter [missing_resource: filtercode.asm][Addressing Modes for TI TMS320C55x](#).

filtercode.asm

```
.ARMS_off                                     ;enable
assembler for ARMS=0
    .CPL_on                                   ;enable
assembler for CPL=1
    .mmregs                                  ;enable mem
mapped register names

    .global _filter
    .global _inPtr
    .global _outPtr

    .copy "macro.asm"                        ; Copy
in macro declaration

    .sect ".data"

FIR_len1      .set 8                         ; This
is a 8-tap filter

    .align 32                                ; Align
to a multiple of 16
coef1                                                 ;
assign label "coef1"
    .copy "coef.asm"                         ; Copy
in coefficients

    .align 32
inputBuffer    .space 16*FIR_len1             ;
Allocate 8 words of storage for filter state

new_sample_index
```

Allocate storage to save index in inputBuffer

```
.word 0
```

```
.copy "testvect.asm"
```

```
.sect ".text2"
```

_filter

```
ENTER_ASM ; Call  
macro. Prepares registers for assembly
```

```
MOV #0, AC0 ; Clears AC0  
and XAR3
```

```
MOV AC0, XAR3 ; XAR3  
needs to be cleared due to a bug
```

```
MOV dbl (*(_inPtr)), XAR6 ; XAR6 contains  
address to input
```

```
MOV dbl (*(_outPtr)), XAR7 ; AR7 contains  
address to output
```

```
BSET AR2LC ; sets  
circular addressing for AR2
```

```
MOV #inputBuffer, AR2 ; State  
pointer is in AR2
```

```
MOV mmap(AR2), BSA23 ; BSA23  
contains address of inputBuffer
```

```
MOV #new_sample_index, AR4 ; State index  
pointer is in AR4
```

```
MOV *AR4, AR2 ; AR2  
contains the index of oldest state
```

```
MOV #coef1, AR1 ;  
initialize coefficient pointer
```

```
MOV #FIR_len1, BK03 ; initialize  
circular buffer length for register 0-3
```

```
MOV *AR6+ << #16, AC0 ;  
Receive ch1 into AC0 accumulator
```

```
MOV AC0, AC1 ;  
Transfer AC0 into AC1 for safekeeping
```

```
MOV HI(AC0), *AR2+ ; store current  
input into state buffer
```

```
MOV #0, AC0 ; Clear AC0  
RPT #FIR_len1-1 ;
```

```

Repeat next instruction FIR_len1 times
    MACM          *AR1+, *AR2+, AC0, AC0          ;
multiply coef. by state & accumulate
    round        AC0                             ; Round
off value in 'AC0' to 16 bits

    MOV          HI(AC0), *AR7+                   ; Store filter
output (from AC0) into ch1
    MOV          HI(AC1), *AR7+                   ; Store saved
input (from AC1) into ch2
    MOV          HI(AC0), *AR7+
    MOV          HI(AC1), *AR7+

    MOV          AR2, *AR4                         ; Save
the index of the oldest state back into new_sample_index
    LEAVE_ASM                                       ; Call
macro to restore registers

    RET

```

`filtercode.asm` applies an FIR filter to the signal from input channel 1 and sends the resulting output to output channel 1. It also sends the original signal to output channel 2.

First, use the MATLAB command `firpm` to generate a 20-tap FIR filter. Type `doc firpm` for information on how to use this command. The filter should pass signals from 4 kHz to 8 kHz. Allow a 1 kHz transition band on each edge of the filter passband. Remember to convert these band edges to digital frequencies based on the 48 kHz sample rate of the system.

Use the `save_coef` command to save the filter. (Make sure you reverse the vector of filter coefficients before you save them.) Also save your filter as a MATLAB matrix, since you will need them later to generate test vectors. This can be done using the MATLAB `save` command. Once this is done, use the `freqz` command to plot the frequency response of the filter.

Part 1: Single-Channel FIR Filter

In this section, you will implement the 20-tap FIR filter. Edit `filtercode.asm` to use the coefficients for this filter by making several changes.

First, the length of the FIR filter for this exercise is 20, not 8. Therefore, you need to change `FIR_len1` to 20. `FIR_len1` is set using the `.set` directive, which assigns a number to a symbolic name. You will need to change this to `FIR_len1 .set 20`.

Second, you will need to ensure that the `.copy` directive brings in the correct coefficients. Change the filename to point to the file that contains the coefficients for your first filter.

Third, you will need to modify the `.align` and `.space` directives appropriately. The TI TMS320C55x DSP requires that circular buffers, which are used for the FIR filter coefficient and state buffers, be aligned so that they begin at an address that is a multiple of a power of two greater than the length of the buffer. Since you are using a 20-tap filter (which uses 20-element state and coefficient buffers), the next greater power of two is 32. Therefore, you will need to align both the state and coefficient buffers to an address that is a multiple of 32. (16-element buffers would also require alignment to a multiple of 32.) This is done with the `.align` command. In addition, memory must be reserved for the state buffer. This is done using the `.space` directive, which takes as its input the number of **bits** of space to allocate. Therefore, to allocate 20 words of storage, use the directive `.space 16*20` as shown below:

```
1          .align 32                % Align to a
multiple of 32
2  coef1   .copy  "coef1.asm"      % Copy FIR
filter coefficients
3
4          .align 32                % Align to
a multiple of 32
5  inputBuffer .space 16*20        %
Allocate 20 words of data space
```

Assemble your code, load the output file, and run. Ensure that it has the correct frequency response. After you have verified that this code works properly, proceed to the next step.

Part 2: Assembly Function Calls From C

So far you have been working exclusively in your `filtercode.asm` file, where the FIR filtering is taking place. In this part, you will be exposed to some of the C code that is required to setup the hardware peripherals. Your goal will be to write C code to change how the filtered output and raw input are sent to the output channels.

You may have noticed that your assembly code seems to automatically run every time a new input sample is ready to be processed. How does the system know to run the assembly routine when new samples are waiting? The answer lies in an **interrupt**, a signal sent by the hardware alerting the processor that new samples are ready to be processed.

Open `main.c`, and find the function named `HWI_RINT0`. This is the function that is called each time the DSP receives a hardware interrupt, signaling the presence of new input samples. You can see that `input[0]` and `input[1]` receive the samples from the four input channels, and then `filter()` is called, beginning your assembly routine in `filtercode.asm`. After the assembly function returns back into the C code, `output[0]` and `output[1]` hold your four output samples.

Note: The `output[0]` variable is a 32-bit integer. Channel 1 and 2 outputs are expected in the top 16 bits and bottom 16 bits, respectively. Likewise, channels 3 and 4 are expected in the top and bottom 16 bits of `output[1]`.

Now that you understand how your assembly routine is called from C, modify it to return the value of the filter output, instead of writing it to

outBuffer directly in assembly. Modify `main.c` with the following:

1. Replace the function declaration `extern void filter(void)` with `extern int filter(void)`.
2. Create an output variable called `outval` to store the value returned from `filter()`.
3. Store `outval` into output channels 1 and 3, and your unfiltered input sample from channel 1 into output channels 2 and 4.

Note: Use typecasting, bitshifting, and bitwise operations to pack the output variables accordingly.

Now that the C code has been changed, you must modify the assembly code to actually return the value. To do this, there is an established convention for how to pass and return values between C and assembly. The rules for this convention are given in **Section 6.4** of the [TMS320C55x Optimizing C/C++ Compiler User's Guide](#).

Note: Please read and understand this section in the manual before proceeding!

Currently, the filter output and raw input are copied to the `output` buffers:

```
MOV HI(AC0), *AR7+
MOV HI(AC1), *AR7+
MOV HI(AC0), *AR7+
MOV HI(AC1), *AR7+
```

Replace these commands with a single command to return only the filter output. **Hint:** you will need register **T0**.

Now after compiling, loading, and running your code, your filter should behave just as in Part 1. In this second part of the lab, you have learned how to make a call to an assembly routine much more modular as you should know how to pass and return values between C and assembly. This will become valuable in later labs, when you may want to cascade multiple assembly subroutines together.

Part 3: Alternative Single-Channel FIR Implementation

An alternative method of implementing symmetric FIR filters uses the **firsadd** instruction. First, make a copy of **filtercode.asm**, as you will have to demo this part separately from the previous two. Modify your code to implement the filter with a 4 kHz to 8 kHz passband using the **firsadd**.

Two differences in implementation between your code from [Part 1](#) and the code you will write for this part are that **firsadd** requires the states to be broken up into two separate circular buffers. Refer to the **firsadd** instruction on *page 5-152* in the [Mnemonic Instruction Set](#) manual.

```

1      mov      *AR1, *AR2-
; write x(-N/2) over x(-N)
2      mov      HI(AC0), *AR1
; write x(0) over x(-N/2)
3      add      *AR1-, *AR2-, AC0
; add x(0) and x(-(N-1))
4
; (prepare for first multiply)
5      rpt      #(FIR_len1/2-1)
6      firsadd  *AR1-, *AR2-, *CDP+, AC0,
AC1
7      round   AC1
```

```

      8      amar      ?????????????????
; Fill in these two instructions
      9      amar      ?????
; They modify AR1 and AR2
     10
     11
; note that the result is now in the
     12
; AC1 accumulator

```

Because states and coefficients are now treated differently than in your previous FIR implementation, you will need to modify the pointer initializations to

```

      1      bset      AR1LC
; sets circular addressing for AR1
      2      bset      AR2LC
; sets circular addressing for AR2
      3
      4
      5      mov      #firState1, AR1
      6      mov      #firState1Index, AR4
      7      mov      mmap(AR1), BSA01
      8      mov      *AR4, AR1
; get pointer to oldest delayBuf in AR1
      9
     10      mov      #firState2, AR2
     11      mov      #firState2Index, AR5
     12      mov      mmap(AR2), BSA23
     13      mov      *AR5, AR2
     14
     15
     16      mov      #(FIR_len1/2), BKC
     17      mov      #(FIR_len1/2), BK03
; initialize circular buffer length for register

```

0-3

```
18      mov      #coef1, CDP
; CDP contains address of coefficients
19      mov      *AR6 << #16, AC0
; copy input into AC0
```

There are also a couple other changes that need to be made before the code will compile successfully. Read the comments carefully and understand how the **firsadd** instruction works to make the necessary changes. Hint: Make sure accumulator usage (AC0, AC1, AC2) and what is sent to output is correct.

Using the techniques introduced in [Lab 0: DSP Hardware Introduction](#), generate an appropriate test vector and find the expected output in MATLAB. In MATLAB, plot the expected and actual outputs of the filter, and the difference between the expected and actual outputs. Why is the output from the DSP system not exactly the same as the output from MATLAB?

Next, compare the output of this code against the output of the same filter implemented using the **mac** instruction. Are the results the same? Why or why not? Ensure that the filtered output is sent to output channel 1, and that the unmodified output is still sent to output channel 2.

Note: You will lose credit if the unmodified output is not present or if the channels are reversed!

Quiz Information

The points for Lab 1 are broken down as follows:

- 1 point: Prelab (must be ready to show the TA the week before the quiz)
- 3 points: Working code from Parts 2 and 3: you must demonstrate that your code works using input from the function generator and that it works using input from appropriate test vectors. Have an `.asm` file **ready** to demonstrate each.
- 4 points: Quiz score.

The quiz may cover signal processing material relating to fixed point processing fundamentals, convolution, and the differences between ideal FIR filters and realizable FIR filters. You may also be asked questions about digital sampling theory, including, but not limited to, the Nyquist sampling theorem and the relationship between the analog frequency spectrum and the digital frequency spectrum of a continuous-time signal that has been sampled.

The quiz **will** cover the code that you have written during the lab. You are expected to understand, in detail, all of the code in the files you have worked on, even if your partner wrote it. The quiz may cover various key lines of code, 2's complement fractional arithmetic, circular buffers, alignment, typecasting and bit manipulation in C, function calling conventions between C and assembly, and the mechanics of either of the two FIR filter implementations.

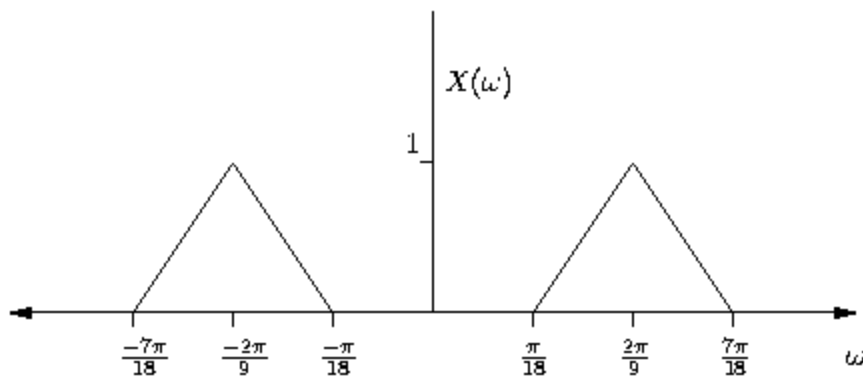
Use the TI documentation, specifically the [Mnemonic Instruction Set](#) manual. Also, feel free to ask the TAs to help explain the code that you have been given.

Lab 2: Prelab part I (multirate theory)

You will work through an example problem that explores the effects of sample-rate compression and expansion on the spectrum of a signal.

Multirate Theory Exercise

Consider a sampled signal with the DTFT $X(\omega)$ shown in [\[link\]](#).



DTFT of the input signal.

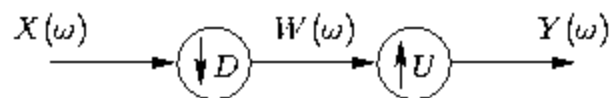
Assuming $U = D = 3$, use the relations between the DTFT of a signal before and after sample-rate compression and expansion ([\[link\]](#) and [\[link\]](#)) to sketch the DTFT response of the signal as it passes through the multirate system of [\[link\]](#) (without any filtering). Include both the intermediate response $W(\omega)$ and the final response $Y(\omega)$. It is important to be aware that the translation from digital frequency ω to analog frequency depends on the sampling rate. Therefore, the conversion is different for $X(\omega)$ and $W(\omega)$.

Equation:

$$W(\omega) = \frac{1}{D} \sum_{k=0}^{D-1} X\left(\frac{\omega + 2\pi k}{D}\right)$$

Equation:

$$Y(\omega) = W(U\omega)$$



Multirate System

Lab 2 : Prelab part II (upsampling and downsampling)

Upsampling and downsampling in Matlab can be implemented by the functions **downsample** and **upsample**. Design a lowpass filter of length 100 and cut off frequency of $\frac{\pi}{4}$ (you can use the Matlab function **fir1**). To visualize the effect of downsampling and upsampling, you might want to plot different magnitude responses in one figure. The Matlab syntax: **[h,w] = freqz(hfilt)** returns the frequency response of the filter **hfilt** without logarithmic scaling (see Matlab help documentation for more information).

Plot the frequency response of your newly designed filter together with its downsampled and upsampled versions by 2 in one figure. Redo the problem with the upsampling and downsampling factors equal to 3. Do your results agree with the theory?

Lab 2: Multirate Filtering

You will implement a multirate system that includes three finite impulse response filters.

Part 1: Filter design using Matlab

Using the zero-placement method, design the FIR filters for the multirate system in [Lab 2: Overview](#). Recall that the z -transform of a length- N FIR filter is a polynomial in z^{-1} , and that this polynomial can be factored into $N - 1$ roots.

Equation:

$$\begin{aligned} H(z) &= h_0 + h_1 z^{-1} + h_2 z^{-2} + \dots \\ &= (z_1 - z^{-1}) (z_2 - z^{-1}) (z_3 - z^{-1}) \dots \end{aligned}$$

Use this relation to design a low-pass filter (for the anti-aliasing and anti-imaging filters of the multirate system) by placing twelve complex zeros on the unit circle at $\pm(\frac{3\pi}{8})$, $\pm(\frac{\pi}{2})$, $\pm(\frac{5\pi}{8})$, $\pm(\frac{3\pi}{4})$, $\pm(\frac{7\pi}{8})$, and $\pm(\pi)$. This filter that you have just designed will serve for both FIR 1 and FIR 3. For filter FIR 2 (operating at the decimated rate), use four equally-spaced zeros on the unit circle located at $\pm(\frac{\pi}{4})$ and $\pm(\frac{3\pi}{4})$. Be sure to adjust the resulting filter coefficients to ensure that the gain does not exceed one at any frequency.

Design your filters by writing a MATLAB script to compute the filter coefficients from the given zero locations. The MATLAB function **poly** is very useful for this; type **help poly** in MATLAB for details.

Once you have determined the coefficients of the filters, use MATLAB function **freqz** to plot the frequency responses. You will find that the frequency response of these filters has a large gain. Adjust the resulting filter coefficients to ensure that the largest frequency gain is less than or equal to one by dividing the coefficients by an appropriate value. Do the frequency responses match your expectations based on the locations of the zeros in the z -plane?

Part 2: Modular Functions by Mixing C and ASM

In [Part 2 of Lab 1](#), you learned how to return a value from assembly to C. High-level C program is easy to develop and maintain, while assembly program directly exploits specialized instructions, providing run-time efficiency.

In this section, you will learn how to pass in an argument from C to the assembly function. More specifically, in the cascaded multirate system, the output of one filtering or rate conversion block is the input to another block. Thus the C program needs to be able to pass the argument to the assembly function. Your new function declaration should be of the following form:

```
extern int filter(int filt_input);
```

To do this, review **Sections 6.4 and 6.5** of the [TMS320C55x Optimizing C/C++ Compiler User's Guide](#). Pay attention to the naming convention, data types and passing arguments order.

Part 3: Multirate System Implementation on the DSP

Implement the complete multirate system shown in [Lab 2: Overview](#). Here are some guidelines:

- First, implement a system that cascades only FIR 1 and FIR 2; exclude the sample-rate compressor, expander, and FIR3. Verify that the response of this two-filter system is as expected. The filters should be written in assembly (as in Lab 1), but the cascading can be done in C.
- Next, implement the entire multirate system. Use a counter to implement the sample-rate compressor and expander. That is, the counter will determine when the compressed-rate processing is to occur, and it can also be used to determine when to insert zeros into FIR 3 to implement the sample-rate expander. This control flow can be written in C.
- At first, use fixed compression and expansion factors of $D = U = 4$. After you have verified that the multirate system works at a fixed rate, you should modify your code so that the rate can be changed easily.

You must be able to quickly change the compression and expansion factors when you demo your code.

Lab 3: Prelab (Part 1)

You will derive the transfer function of a second-order, Direct Form II, infinite impulse response (IIR) filter. Then you will create a fourth-order IIR filter, plot its frequency response, and decompose the fourth-order filter into two second-order sections, choosing an appropriate gain for each stage to prevent overflow.

The transfer function for the second-order section shown in [Lab 3: IIR Filters Overview](#) is

Equation:

$$H(z) = G \frac{1 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

Exercise

First, derive the above transfer function from the block diagram. Begin by writing the **difference equations** for $w[n]$ in terms of the input and past values ($w[n-1]$ and $w[n-2]$). Then write the difference equation for $y[n]$ also in terms of the past samples of $w[n]$. After finding the two difference equations, compute the corresponding Z-transforms and use the relation $H(z) = \frac{Y(z)}{X(z)} = \frac{Y(z)W(z)}{W(z)X(z)}$ to verify the IIR transfer function in [\[link\]](#).

Next, design the coefficients for a fourth-order filter implemented as the cascade of two bi-quad sections. Write a MATLAB script to compute the coefficients. Begin by designing the fourth-order filter and checking the response using the MATLAB commands

```
[B,A] = ellip(4, .25, 10, .25)
freqz(B,A)
```

Note: MATLAB's `freqz` command displays the frequency responses of IIR filters and FIR filters. For more information about this, type `doc freqz`. Be sure to look at MATLAB's definition of the transfer function.

Note: If you use the `freqz` command as shown above, without passing its returned data to another function, both the magnitude (in decibels) and the phase of the response will be shown.

Next you must find the roots of the numerator, **zeros**, and roots of the denominator, **poles**, so that you can group them to create two second-order sections. The MATLAB commands `roots` and `poly` will be useful for this task. Save the scripts you use to decompose your filter into second-order sections; they will probably be useful later.

Once you have obtained the coefficients for each of your two second-order sections, you are ready to choose a **gain** factor, G , for each section. As part of your MATLAB script, use `freqz` to compute the response $\frac{W(z)}{X(z)}$ with $G = 1$ for each of the sets of second-order coefficients. Recall that on the DSP we do not represent numbers greater than or equal to 1.0. If the maximum value of $\left| \frac{W(z)}{X(z)} \right|$ is or exceeds 1.0, an input with magnitude less than one could produce $w[n]$ terms with magnitude greater than or equal to one; this is **overflow**. You must therefore select a gain values for each second-order section such that the response from the input to the states, $\frac{W(z)}{X(z)}$, is always less than one in magnitude. In other words, set the value of G to ensure that $\left| \frac{W(z)}{X(z)} \right| < 1$.

After finishing Part 1, move on to [Lab 3: Prelab \(Part 2\)](#), where you explore and learn how to mitigate the effects of quantization.

Lab 3: Prelab (Part 2)

You will design a fourth-order notch filter and investigate the effects of filter-coefficient quantization. You will compare the response of the filter having unquantized coefficients with that of a filter having coefficients quantized as a single, fourth-order stage and with that of a filter having coefficients quantized as a cascade of two, second-order stages.

Filter-Coefficient Quantization

One important issue that must be considered when IIR filters are implemented on a fixed-point processor is that the filter coefficients that are actually used are quantized from the "exact" (high-precision floating point) values computed by MATLAB. Although quantization was not a concern when we worked with FIR filters, it can cause significant deviations from the expected response of an IIR filter.

By default, MATLAB uses 64-bit floating point numbers in all of its computation. These floating point numbers can typically represent 15-16 digits of precision, far more than the DSP can represent internally. For this reason, when creating filters in MATLAB, we can generally regard the precision as "infinite," because it is high enough for any reasonable task.

Note: Not all IIR filters are necessarily "reasonable"!

The DSP, on the other hand, operates using 16-bit fixed-point numbers in the range of -1.0 to $1.0 - 2^{-15}$. This gives the DSP only 4-5 digits of precision and only if the input is properly scaled to occupy the full range from -1 to 1 .

For this section exercise, you will examine how this difference in precision affects a **notch filter** generated using the **butter** command: `[B,A] = butter(2, [0.07 0.10], 'stop')`.

Quantizing coefficients in MATLAB

It is not difficult to use MATLAB to **quantize** the filter coefficients to the 16-bit precision used on the DSP. To do this, first take each vector of filter coefficients (that is, the **A** and **B** vectors) and divide by the smallest power of two such that the resulting absolute value of the largest filter coefficient is less than or equal to one. This is an easy but fairly reasonable approximation of how numbers outside the range of -1 to 1 are actually handled on the DSP.

Next, quantize the resulting vectors to 16 bits of precision by first multiplying them by $2^{15} = 32768$, rounding to the nearest integer (use **round**), and then dividing the resulting vectors by 32768. Then multiply the resulting numbers, which will be in the range of -1 to 1, back by the power of two that you divided out.

Effects of quantization

Explore the effects of quantization by quantizing the filter coefficients for the notch filter. Use the **freqz** command to compare the response of the unquantized filter with two quantized versions: first, quantize the entire fourth-order filter at once, and second, quantize the second-order ("bi-quad") sections separately and recombine the resulting quantized sections using the **conv** function. Compare the response of the unquantized filter and the two quantized versions. Which one is "better?" Why do we always implement IIR filters using second-order sections instead of implementing fourth (or higher) order filters directly?

Be sure to create graphs showing the difference between the filter responses of the unquantized notch filter, the notch filter quantized as a single fourth-order section, and the notch filter quantized as two second-order sections. Save the MATLAB code you use to generate these graphs, and be prepared to reproduce and explain the graphs as part of your demo. Make sure that in your comparisons, you rescale the resulting filters to ensure that the response is unity (one) at frequencies far outside the notch.

Lab 3: Lab

You will implement a fourth-order, elliptical, low-pass infinite impulse-response (IIR) filter as a cascade of two second-order sections.

Overview

In this lab, you implement a fourth-order IIR filter completely in fixed-point C. While programming in C provides ease of coding, portability, and comprehension, fixed-point processing raises a few challenges that were handled automatically in assembly. In particular, it is the programmer's responsibility to explicitly handle overflow errors and accumulator sizing.

On the DSP, you will *write and test* the C function for the elliptic low-pass filter designed from [Prelab \(Part 1\)](#). You should not try to implement the notch filter designed in [Prelab \(Part 2\)](#), because it will not work correctly when implemented using Direct Form II. (Why not?)

To implement the fourth-order filter, start with a single set of second-order coefficients and implement a single second-order section. A suggested outline of the implementation steps are:

1. On paper, design the algorithm for a modular implementation of a single second-order section.
2. Write the algorithm in C, handling truncation, overflow, saturation, and accumulator sizing.
3. Verify functionality of this single bi-quad using the frequency sweep test-vector, and the function generator/oscilloscope.
4. In Matlab, pair the poles and zeros to maximize the gain factors for each section, and on the DSP, verify the correct operation of each bi-quad independently.
5. Finally, write C code to implement the cascade. The modular design of the second-order section should convince you of the benefits of C programming.

Part 1: Design on Paper

The first step, and the majority of the work, is to implement a single second-order section, which was shown in [Figure 1](#). Before writing in C, carefully design and plan out the algorithm on paper in pseudo-code. For an example of how pseudo-code is used to implement an FIR filter, see this link [FIR filter implementation](#).

From your design, you should have a very clear idea about:

- the chronological order of how the intermediate states $\{w[n], w[n-1], w[n-2]\}$, and the output, $y[n]$ should be updated.
- how pointers or data should move after sample $x[n]$ has been processed, but before $x[n+1]$ comes in.
- the data types required for all buffers, accumulators, and temporary variables you may need.

Exercise:

Problem:

Which buffer should be circularly addressed: coefficients or state buffer?

Solution:

The intermediate state buffer. If you do not clearly understand why, go back to Figure 1 and spend more time on this part of the lab!

Part 2: C Implementation of a Second-Order Section

You may want to implement the second-order section with the following function declaration:

```
long iirSoS(int *b, int *a, int
*w_states, long input);
```

The above declaration is only a recommendation, and the exact number of arguments or even the datatypes used can be designed differently. The point

though, is that this specification enables function re-use, unlike Lab 2 where different assembly functions were written for each filter.

In the suggested function declaration above, the first two arguments are pointers to the filter coefficients, the third argument is a pointer to the intermediate state buffer, and the final argument is the current input sample. The returned value is the output of the given second-order section.

Note: Why would one want to declare the input as type

long

?

Here are some guidelines for implementing the second-order section in C:

- make sure you size accumulators appropriately, and type cast accordingly when accumulating values.
- for a naive initial implementation of the circular buffer, let the newest element, $w[n]$ always have the lowest array index (i.e., $w_states[0] = w[n]$). Copy over other values as required; order matters!
- Handle overflow errors correctly. You may find it convenient to write a **helper function** that handles overflow correctly. The function declaration for this may look like:

```
int long2int( long );
```

See the Troubleshooting section below for tips on how to correctly handle overflow issues.

Finally, verify that the second-order section works correctly when using both sets of the second-order coefficients. To do this:

1. In Matlab, obtain your filter coefficients by properly grouping the pole/zero pairs. A given combination will result in a pair of scaling

- factors (G_1, G_2), where G is as defined in Figure 1. The rule of thumb is that you want to pick the pole/zero pairs such that the *worst-case* gain, or $\min(G_1, G_2)$ is as close to 1 as possible.
2. From the Prelab exercise, verify that the quantized coefficients are properly scaled and acceptable.
 3. In C, create buffers for each set of coefficients and intermediate states.
 4. Initialize the intermediate state buffers.
 5. Generate a test vector and verify your implementation. Refer to [Step 9 of Lab 0](#).

Part 3: Cascade Implementation

Once your single second-order IIR section is working properly, you can proceed to implement the cascade of second-order sections. The modular design in Part 2 should make this fairly straightforward. Make sure to apply the gain factors to the two filter inputs. Type-casting and shifting may be required!

Fixed-Point Processing: Troubleshooting

This section contains additional information that will help you avoid common pitfalls associated with fixed-point processing.

Coefficients greater than 1:

You may have noticed that some of the coefficients you have computed for the second-order sections are larger than 1.0 in magnitude. For any stable second-order IIR section, the magnitude of the "0" and "2" coefficients (a_0 and a_2 for example) will always be less than or equal to 1.0 (make sure you understand why!). However, the magnitude of the "1" coefficient can be as large as 2.0. To overcome this problem, you will have to divide any coefficient larger than 1 by two prior to saving them for your DSP code. Then, in your implementation, **accumulate twice** to compensate for using half the coefficient value.

Handling overflow:

Overflow is really only a problem when one needs to *truncate* the result of an accumulation (i.e., store a 16-bit number into a buffer).

When accumulating numbers in twos-complement notation, a nice property is that the final value will be correct even if intermediate values overflow, *as long as the final accumulated value is in the range of representable numbers* (i.e., in between -32768 and +32767).

If the final value is outside of this range, then one solution is to *saturate* the value to +32767 or -32768. See [Fixed-Point Quantization](#) for more information about the different errors incurred by fixed-point processing.

Extra credit

In Part 2, we proposed an inefficient (yet simple) implementation of a circular buffer, where data elements are shifted one by one. This is clearly less efficient than circular addressing, where only a single pointer moves through the buffer and data elements remain fixed. **For an extra credit point**, write C code to get the compiler to implement circular addressing. You must be able to explain your C code and show the circular addressing in the assembler output. **Hint:** you will need an additional argument into `iir_SoS()` that keeps track of the current sample index.

Grading

Your grade on this lab will be split into three parts:

- 1 point: Prelab. Be prepared to show your Matlab code used to study coefficient quantization, and to compute poles and zeros.
- 3 points: Code. Your DSP code implementing the fourth-order IIR filter.
- 4 points: Written quiz. The quiz may cover differences between FIR and IIR filters, the prelab material, errors induced by fixed-point processing, and the MATLAB exercise.
- 1 point extra credit: Implementing hardware circular addressing in C; must verify using assembler output.

Lab 4 Prelab 4

You will investigate the effects of windowing and zero-padding on the Discrete Fourier Transform of a signal, as well as the effects of data-set quantities and weighting windows used in Power Spectral Density estimation.

MATLAB Exercise, Part 1

Since the DFT is a sampled version of the spectrum of a digital signal, it has certain sampling effects. To explore these sampling effects more thoroughly, we consider the effect of multiplying the time signal by different window functions and the effect of using zero-padding to increase the length (and thus the number of sample points) of the DFT. Using the following MATLAB script as an example, plot the squared-magnitude response of the following test cases over the digital frequencies

$$\omega_c = \left[\frac{\pi}{8}, \frac{3\pi}{8} \right].$$

1. rectangular window with no zero-padding
2. hamming window with no zero-padding
3. rectangular window with zero-padding by factor of four (i.e., 1024-point FFT)
4. hamming window window with zero-padding by factor of four

Window sequences can be generated in MATLAB by using the `boxcar` and `hamming` functions.

```
1  N = 256;                                % length of
test signals
2  num_freqs = 100;                        % number of
frequencies to test
3
4  % Generate vector of frequencies to
test
5
6  omega = pi/8 + [0:num_freqs-
```

```

1]'/num_freqs*pi/4;
7
8 S = zeros(N,num_freqs);
% matrix to hold FFT results
9
10
11 for i=1:length(omega)
% loop through freq. vector
12     s = sin(omega(i)*[0:N-1]');
% generate test sine wave
13     win = boxcar(N);
% use rectangular window
14     s = s.*win;
% multiply input by window
15     S(:,i) = (abs(fft(s))).^2;
% generate magnitude of FFT
16
% and store as a column of S
17 end
18
19 clf;
20 plot(S);
% plot all spectra on same graph
21

```

Make sure you understand what every line in the script does. What signals are plotted?

You should be able to describe the tradeoff between mainlobe width and sidelobe behavior for the various window functions. Does zero-padding increase frequency resolution? Are we getting something for free? What is the relationship between the DFT, $X[k]$, and the DTFT, $X(\omega)$, of a sequence $x[n]$?

MATLAB exercise, Part 2

In this section, you will resolve the two closely spaced sine waves using a Fourier transform method. Consider the signal:

Equation:

$$x(n) = \sin(2\pi f_1 n) + \sin(2\pi f_2 n)$$

consisting of two sine waves of frequency 2000 Hz and 2100 Hz with sampling frequency of 8000 Hz. Here, n is the discrete time index.

Generate a block of 256 samples of $x(n)$ and use the Fast Fourier Transform (fft) command to determine the two frequency components.

Exercise:

Problem:

What is the closest frequency to 2000 Hz that you can resolve using the Fourier transform method? Which of the following method applied to $x(n)$ results in the best resolving capabilities? Why?

- rectangular window with no zero-padding
- hamming window with no zero-padding
- rectangular window with zero-padding by factor of four (i.e., 1024-point FFT)
- hamming window window with zero-padding by factor of four.

MATLAB exercise Part 3

Example:

The following Matlab code can be used to generate a pure tone:

```
fs = 8000; %sampling rate
duration = 1; % sec
t = linspace(0, duration, duration * fs); % time
axis
```

```
freq = 600; % Hz  
x = sin(2*pi*freq*t);
```

To listen to it, you can use:

```
soundsc(x, fs)
```

This will be useful for testing purposes in Lab 4.

Short-time spectral analysis is an important technique that is used to visualize the time evolution of the frequency content of non-stationary signals, such as speech. The fundamental assumption is that the signal is modeled as being *quasi-stationary* over short time periods; in many speech applications, this period is on the order of 20-30 milliseconds.

The short-time Fourier transform (STFT) is defined to be:

Equation: $X(\tau, f) = \text{FT}\{x(t)w(t-\tau)\}$

where f is frequency, FT represents the Fourier transform operation, and $w(t-\tau)$ is a window with finite-time support centered at time τ (i.e., it is non-zero for only a short amount of time near τ).

In the case of digital time and frequency, the rate at which τ is evaluated, along with the block size used to compute the FFT determines the amount of data overlap that occurs in evaluating the STFT over time.

The **spectrogram** is just the magnitude-squared of the STFT, and can be computed in MATLAB:

```
spectrogram(x, nwin, noverlap);
```

which computes the STFT using a Hamming window of length $nwin$, every **nwin-noverlap** samples. The question of how much to overlap can be understood by asking how many time/frequency samples are required to *fully* represent the continuous time-frequency STFT; for the interested reader, see the section titled "Analysis of Short Term Spectra" in [\[Allen77\]](#).

Exercise:

Problem:

Plot the spectrogram of the first two signals you generate in part 2 with no overlap and 50% overlap. How are the spectrograms different between the two methods?

For students in the Wednesday and Thursday sections: repeat this for the the following frequency-sweep signal: `S =`

```
chirp(0:1/8000:0.5,1000,0.5,5000);
```

Type `doc chirp` to understand the input arguments to the chirp function. What is going on at 0.4 seconds into the signal?

Lab 4: Lab

Implement a spectrogram application in the Native Android environment. Signal processing includes windowing, zero-padding, overlapping, FFT, and working with complex numbers.

Lab Overview

In this lab, you will create an Android application that plots the spectrogram of streaming audio, and deploy it on the [Google Nexus 7](#) tablet.

Similar to the previous labs, you will be provided with an existing project with all of the required peripherals already set up. Unlike the previous labs where data was processed sample-by-sample, this lab requires block-based processing.

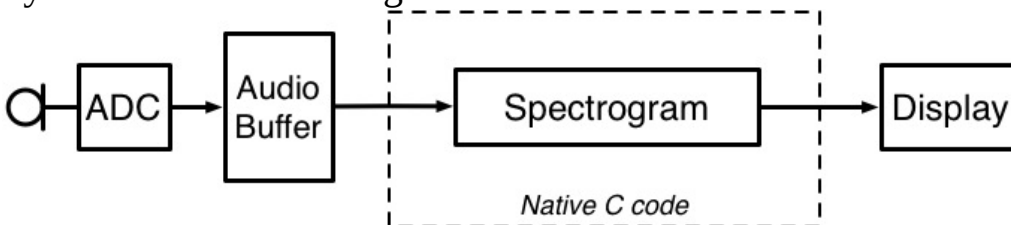
Exercise:

Problem:

What is the *minimum* latency for processing a block of length 256, with an audio sampling rate of 8 kHz?

[\[link\]](#) shows a block diagram of the system you will be implementing. The project is already configured to stream audio and display the processed output to the screen.

System-level Block Diagram



You will be focusing on the signal processing tasks, which will be implemented in [native C](#) code. For those that are interested in the Android Java specifics, we will provide an optional tutorial that shows you how to build the project from scratch, explains the different Android classes that were used, and along the way, provides supplemental links to references and other useful Android tutorials available on the web.

Part 1: Getting Started with Android and Eclipse

In this section, you first setup your Android device in development mode, import a skeleton project into Eclipse, and familiarize yourself with the Android project structure and build process.

Setting Up the Google Nexus for Development

On the tablet, you must enable the **Developer options** under **Settings**:

1. Go to **Settings**,
2. Click on **About tablet**,
3. Click **Build number** seven times (yes, 7).

Under **Settings > Developer options**, enable the **Stay awake** and **USB debugging** options.

Setting Up Eclipse

The development environment you will be using is the [Nvidia Tegra Android Development Pack 2.0](#). To get started:

1. Start Cygwin by double-clicking on **C:\NvPack\cygwin\cygwin.bat**
2. In the prompt, navigate to the Eclipse folder by typing **cd /cygdrive/c/NvPack/eclipse**
3. Launch Eclipse with **./eclipse**

Note: [Cygwin](#) provides a Linux-like environment for Windows, and launching Eclipse from within Cygwin is required for native C debugging to work correctly.

Choose a new workspace on your **U:** drive, similar to what you did in Lab 0. This step feels familiar because CCS is actually based on the Eclipse framework.

Importing the Project

Once Eclipse opens, select **File > Import...**

1. General > Existing Projects into Workspace
2. V:\ece420\nexus\Lab4\
3. Check "Copy into Workspace"

Close out of the "Welcome" screen.

Once the project is imported, it will try to build automatically. If you see a build error, you will need to define the \$NDKROOT variable in your workspace:

1. Go to **Project > Properties**
2. Expand **C/C++ Build > Environment**
3. **Add...**
4. **Name:** NDKROOT, **Value:** C:\NvPack\android-ndk-r8

Understanding the Android Project Structure

An Android project with Native code support has 4 main components:

- **.\AndroidManifest.xml** - contains app-related information, such as project name, activities, and required peripherals (e.g., microphone).
- **.\res\layout\main.xml** - describes the layout of the user interface. In our project, we define an **ImageView**, which is used to display the spectrogram.
- **.\src** - contains the Java source files.

- `.\jni\` - contains the native C source and Make files.

Use the **Project Explorer** (left pane in Eclipse) to open the different files and familiarize yourself with each component.

Building the Project

Building the project actually consists of first building your Native C code as a library using **ndk-build**, and then building the Java code, which is responsible for loading the C library and executing your functions.

For this lab project, the build process in Eclipse has already been configured to build everything automatically, and so building your project is as simple as clicking **Project > Build Project**.

Note: You may want to uncheck **Build Automatically**, and then build by first **Cleaning** the project and checking the "Immediately start build" option.

C Syntax Support

Open `.\jni\process.c` in the Eclipse editor. This file contains the processing function, which is where you will implement your signal processing. If you see notifications for syntax errors, then in the **Project Explorer**, right-click on the project name and go to **Project Properties > C/C++ General > Paths and Symbols** and add the following three files into the **Include directories** for **GNU C**:
`C:\NvPack\android-ndk-r8\platforms\android-9\arch-arm\usr\include`
`C:\NvPack\android-ndk-r8\sources\cxx-stl\gnu-libstdc++\include`
`C:\NvPack\android-ndk-r8\sources\cxx-stl\gnu-libstdc++\libs\armeabi-v7a\include`

Part 2: A Development Environment for Signal-Processing Applications

Function Declarations in Java and C

In the `Lab4Activity` Java class, the following function declaration is made:

```
public static native void process(ShortBuffer
inbuf, DoubleBuffer outbuf, int N);
```

Exercise:

Problem: Where is the `Lab4Activity` class defined?

Solution:

According to the *"Understanding the Android Project Structure"* section in Part 1, you can find it in `Lab4Activity.java` in a sub-directory of `.\src\`

Here, the `native` keyword states that `process()` is a Native C function. The first argument points to the Java-equivalent of a `short[N]` array, which holds a block of the 16-bit audio data. The second argument points to the Java-equivalent of a `double[N]` array, which you have to fill with data to visualize on-screen.

In `process.c`, the C-equivalent function declaration for `process()`, along with pointers to the input and output buffers have already been written for you. Appropriate header files have also been included, including the [FFTW](#) library, a popular C library for implementing the FFT.

Debugging in Java

To introduce the Java debugging environment, you will verify that the mechanism for using *test vectors* is working correctly. First, in the `Lab4Activity` class definition, set the variable `FILE_INPUT` to `true`. Set a breakpoint on the line where `process()` is called.

Click `Run > Debug`, and debug as an `Android Application`. The project will automatically rebuild if out-of-date, and should switch to the `Debug Perspective`. The application will automatically run and be halted at the set breakpoint.

Click `Window > Show View > Expressions` and add the following expression: `sb.get(0)`, which returns the *first* value in the buffer. Verify that the value is correct by comparing to the values in the lookup table defined in `LOOKUP.java`. Repeat this for other values in the table to convince yourself that the test vector values were copied correctly.

Note: The [Android Reference](#) guide contains detailed information about the classes you will encounter in Java (e.g., the [ShortBuffer](#) class).

This process has shown you how to set breakpoints and single-step through Java code.

Exercise:

Problem:

What happens if you try to step *into* the `process()` function?

Solution:

The debugger will step *over* the function because it is written in C.

Debugging in C

Debugging in C is done using `ndk-gdb`, which is supported in Eclipse. To do this, first terminate any existing debug session and then on the Android device, exit the current application by hitting the home button.

To configure GDB, switch back to the `C/C++ Perspective` and do the following:

1. Click `Run > Debug Configurations...`
2. Select `Android NDK Application` and press the `New` icon, and rename the configuration to `Lab4 (1)`.
3. On the `Android` tab:
 - a. Under `Project`, hit `Browse...` and select the current project.
 - b. Under `Misc.`, select `Attach to the running application`.
4. On the `Debugger` tab:
 - a. Select the `GDBserver Settings` tab.
 - b. Use the `APK bundled` GDBserver.
5. `Apply` the settings and `Close`.

In order to launch GDB, you must first run the application on the Android device, and then attach GDB to the running application. To do this:

1. Set a breakpoint inside the `for` loop in `process.c`.
2. Click `Run > Run`. You should see the application launch on the Android device, ignoring any Java breakpoints that you have set.
3. After the application starts, click `Run > Debug As... > Android NDK Application`. The `Debug Perspective` should launch again, but this time with `gdbserver debugger`. The processor will halt at the set C breakpoint.

Note: If `Android NDK Application` does not show up, make sure that `Lab 4` is highlighted in the Project Explorer before trying to launch

the debugger.

In the **Expressions** window, the **sb.get(0)** (which is a Java method) will have generated an error as we are now debugging in C. Verify that **inBuf** has the same values as the test vector look-up table by adding **inBuf[0]** to the list of expressions; check several different array indices.

Exporting Variables to a File

A useful feature that is fully supported in CCS is the ability to export processor memory to a file, which can then be imported into MATLAB for further analysis. To enable this feature in Eclipse, the lab machines have the [eVars](#) plugin installed. As an example, to export the **inBuf** array to file:

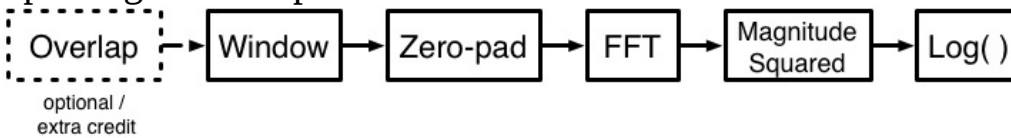
1. In the **Debug Perspective**, go to **Window > Show View > Variables**.
2. Right-click on the **inBuf** pointer and select **Display as Array**.
3. Click the **Expand Variables** icon multiple times until the entire array has been expanded.
4. Once the array is fully expanded, click on the **Export Variables** icon, and save to a txt file.
5. Use [evars2array.m](#) to read in the text file into a MATLAB vector.

In the next lab, we will see how to create and write to a file on the Android device, and use **adb** to download this file to the host machine, directly from MATLAB.

Part 3: A Spectrogram Algorithm in MATLAB

As an initial step towards implementing the spectrogram in Android, you will first implement it in MATLAB. [\[link\]](#) shows the components that your spectrogram algorithm should have; the ability to overlap, while important, will be left for extra credit, and is therefore optional.

Spectrogram Components



Note: If you need to see an existing implementation, type

`open spectrogram`

in the MATLAB prompt. This will open the spectrogram function you used in the Prelab.

When implementing your spectrogram algorithm, make the following assumptions:

- use a Hamming window
- the window length is $N=256$
- zero-pad by a factor of 2
- do not overlap

Here are some things to keep in mind:

- do not vectorize your code or use MATLAB-specific helper functions that are not available on the tablet (such as `zeros()` or `norm()`), as you want to make porting it to C as straightforward as possible.
- Retain only half of the FFT output, as it is conjugate symmetric (make sure you know why!)
- If $X = X_r + jX_i$ is a complex number, the *magnitude squared* operation computes $X_r^2 + X_i^2$.
- Because power can vary by orders of magnitude, the *Log* computation is used to reduce the dynamic range of the spectrogram output; this is useful when visualizing the data.

Exercise:

Problem:

If your input signal is 8192 samples long, then your spectrogram output can be thought of as a 256 x 32 real-valued matrix. Make sure to understand why. You can then use the `image()` or `imagesc()` functions in MATLAB to visualize the data.

Part 4: A C Implementation of the Spectrogram

Specifications

Your task is to implement a C version of the spectrogram algorithm that you wrote in Part 3. Here are some guidelines for how to proceed:

- Remember you are doing block-based processing. Every time `process()` is called, `inBuf` has `N` samples available to be processed.
- Read **Section 2.1** of the [FFTW tutorial](#) to understand the data structures and function calls of the FFTW library.
- Remember that floating point is available on this processor.
- Use the test vector to verify that intermediate operations are being computed correctly (e.g., multiplication, zero-padding, log function, etc.).
- For **extra credit**, implement a scheme that allows for arbitrary overlapping. This may require modifying code in `Lab4Activity.java`

Scaling the Output

The values of `outBuf` must be between 0.0 and 1.0. This is because the output values are directly mapped to RGB colors, with each color channel being 8 bits. As the spectrogram output is generally not in between 0.0 and 1.0, you will need to find an appropriate mapping.

One possible mapping is to linearly scale and saturate the spectrogram output; you must determine the scaling parameters experimentally by processing real audio data. Here is an outline of one way to do this:

- Start up the GDB debugger and **Resume** with all breakpoints disabled.
- While playing a loud tone (i.e., generate in MATLAB and play out through headphones), set a breakpoint right before your **process()** function returns.
- Export the **inBuf** array to a file. Review *Part 2: Exporting Variables to a File* if you don't remember how.
- Repeat this process for noise-only input.
- Import the two files into MATLAB to determine a suitable dynamic range.

Note: This method also enables you to verify the functional correctness of your C code by exporting the spectrogram **output** to a file.

Quiz Information

Be able to describe the effects of windowing and zero-padding on FFT spectral analysis. Know basic properties of the Fourier transform, DTFT, and DFT. What are the trade-offs between block-based and sample-by-sample processing? Although we did not require you to implement it, understand the effects of overlapping when computing the STFT.

Understand the basic Android project structure and the relationship between Java and C programming for Android.

Lab 5: Prelab 5

Histogram equalization and color space conversion

Please download the [eco.tiff](#), [lena.tiff](#) and copy them into your Matlab current workspace before continuing.

Overview

Digital images are made up of picture elements, more commonly known as pixels arranged in a rectangular grid. Three frequently encountered image formats are:

Binary image: are 2D arrays which has only two values 1 or 0 where 1 corresponds to white and 0 corresponds to black.

Intensity or grey scale image: are 2D arrays where pixel intensities have a n-bit representation. For example, an 8-bit image has a variation from 0 to 255 where 0 represents black, 255 represents white and intermediate values correspond to gray levels that span the range from black to white.

RGB color images are 3D arrays that assign three numerical values to each pixel, each value corresponds to the red, green, and blue image channel, respectively.

I. Image enhancement

One simple image enhancement method is increasing the image brightness. Read one of Matlab's default image `cameraman.tif` by using `imread('cameraman.tif')`, then increase its brightness by using the `imageadd` function. To show the image, use the function `imshow`. What is the dynamic range (the number of distinct pixel values in an image) of the original and the enhanced image? Now try to enhance the `eco.tif` image. Could you enhance the quality of the image by simply increasing its brightness?

II. Histogram equalization

Histogram equalization is one of the most commonly used image contrast enhancement technique. The approach is to design a transformation in such a way that the gray values in the output image are uniformly distributed. You'll have a chance to implement histogram equalization in C in lab 4.

For this part, use Matlab built in function `histeq` to perform the histogram equalization on the `eco.tif` , then save the image in Bitmap format to the disk.

Question: Can you improve the result of enhancement by repeating the histogram equalization? Why?

III. Color spaces conversion:

An artist might mix their primary colors on a palette to visualize the color they want to pick. A color space is like a digital palette but a more precisely organized one. Learning to visualize the color space will help you envision the suitable one for your image processing task.

1. We will first look at the popular RGB color space. Load the Lena image into the Matlab workspace. The color image in Matlab is represented as a three dimensional array ($M \times N \times 3$) or $((M \times N \times 3))$ depending on the color model RGB, CMYK, HSL. We will first look at the popular RGB color space. Load the Lena image into Matlab workspace. Zero out two channels and keep one channel intact. Display the result.

Question:What do the three channels represent, and explain how would modifying each of the channels would change the image?

2. Now we investigate a different color space called HSV which is more closely related to our perception of color. We will reuse the Lena image. Convert the Lena image, which is in RGB color space, into HSV color space. To understand the significance of each channel, apply different scaling for each channel as follows. Scale the magnitude of the H channel by 0.1, 0.5 and 0.7. Combine your scaled H with the unscaled S and V channels, transform to RGB image and display it. Repeat the previous step for S and V channel.

Question: What do the three channels represent, and explain how would modifying each of the channels would change the image?

IV. Histogram equalization on color image:

Apply the histogram equalization to the three channels of the Lena image in RGB color space, combine and display the result.

Question: if you are to apply the histogram equalization on HSV image, which channels would you use? Why?

Use your answer to do the histogram equalization in the HSV color space on the Lena image.

Which color spaces should we use when we perform the histogram equalization? Why?

Lab 5: Histogram equalization

Students will implement histogram equalization on the Android platform. Concepts include color conversion, histograms, cumulative distribution functions, and tone mappings.

Lab Overview

In this lab, you will create an Android application that performs histogram equalization on streaming video, and deploy it on the Google Nexus 7 tablet. You will input a video stream from the front-facing camera, equalize each frame of video, and then display the equalized video on the screen alongside the unprocessed video.

Part 1: Setting up Android and Eclipse

After completing Lab 4, you should be familiar with the Android development process, and you should feel comfortable working with Eclipse and the Nexus 7. This includes understanding the difference between Java and native C code, being able to compile and run your code on the tablet, and using both the Java and C debuggers to troubleshoot your code.

Note: If you are not familiar with all of the above concepts, go back to Lab 4 and read the relevant sections.

1. Start Cygwin by double-clicking on `C:\NvPack\cygwin\cygwin.bat`
2. In the prompt, navigate to the Eclipse folder by typing `cd /cygdrive/c/NvPack/eclipse`
3. Launch Eclipse with `./eclipse`

Once Eclipse opens, select `File > Import...`

1. General > Existing Projects into Workspace
2. V:\ece420\nexus\OpenCV - 2.4.5
3. V:\ece420\nexus\Lab5\
4. Check "Copy into Workspace"

Additionally, in Lab5 **Project Properties > Android**, remove the current reference to the OpenCV library, which should have a red check mark next to it, and add the library that is in your workspace.

The lab machines currently do not support Android 4.2.2. If your tablet has upgraded, you have two options:

- Get a TA to roll back your OS to 4.1.2. See this [link](#) for instructions.
- Develop on your own machine by installing the latest Nvidia Tegra Development Pack.

For this lab project, the build process in Eclipse has already been configured to build everything automatically, and so building your project is as simple as clicking **Project > Build Project**.

Note: You may want to uncheck **Build Automatically**, and then build by first **Cleaning** the project and checking the "Immediately start build" option.

The first time you run the application, you will be asked to download the OpenCV Manager from the Play Store; make sure you are connected to the internet.

Running the demo now will only stream video in grayscale. Pressing the **...** in the corner of the screen will bring up the options menu, allowing you to view the RGB or equalized image; you will implement these functionalities in Parts 2 and 3.

Part 2: Color Conversion

In this part, you will implement your own color-conversion algorithm. The purpose is to obtain a better understanding of different color spaces, along with becoming more comfortable with accessing multidimensional data in an environment other than MATLAB.

As discussed in the prelab, there is more than one way to represent the pixels in an image. We will be applying histogram equalization in a color space known as YUV. The Y channel encodes the luma component (brightness), and the U and V are the chroma (color) components. The equalized image will be displayed in RGB.

The pixels received as input from the camera are in the YUV420sp format, which signifies how the data is packed into a linear array. The [Wikipedia article on YUV](#), especially the section titled "**Y'UV420p (and Y'V12 or YV12) to RGB888 conversion**", may be a useful reference; make sure that you thoroughly understand the structure of a single frame of data.

Open `jni_part.cpp` and complete the `YUV2RGB()` function. You will be able to see your results by selecting "Preview RGB" from the options menu.

To learn how to access individual pixel values, read the [OpenCV documentation on basic operations with images](#). It is also important to know that you are working with images that use 8-bit unsigned pixel values.

To assist you in writing the conversion code, you may reference the [YUV to RGB conversion code](#) provided on Wikipedia. Remember, you will need to access the pixel values using the proper matrix syntax. See the [OpenCV matrix documentation](#).

Note: Make sure you understand every single line of the code. We are allowing you to use this (very helpful) resource instead of writing the code from scratch, but you need to fully understand how the conversion is taking place.

After you verify you are streaming color images, continue on to the next part to implement histogram equalization.

Part 2: Histogram Equalization Implementation

In `jni_part.cpp` in the `HistEQ()` function, notice that the lines

```
Mat* pYUV=(Mat*)addrYuv;  
Mat* pRGB=(Mat*)addrRgba;
```

set up pointers to the video input and output. One frame of video lies in the matrix pointed to by `pYUV`. Your assignment is to modify this matrix of pixels so that the left half of the image is histogram equalized, and the right half of the image remains unprocessed. Then you will need to convert the YUV format image to RGB format and save it in the matrix pointed to by `pRGB`. This `HistEQ C` function is called every time a new video frame is ready to be processed, allowing an entire video stream to be processed over time. The algorithm can be broken up into 4 steps:

Compute the histogram of the Y channel

Compute the CDF of the histogram

Apply equalization to the Y channel

Convert the equalized image to RGB

Unless you are already familiar with histogram equalization, reading the [OpenCV histogram equalization tutorial](#) should be helpful in understanding how this algorithm affects an image. OpenCV is an open source computer vision library that provides many useful functions for Android developers to use when creating applications that rely on image and video processing. You will use some of OpenCV's functionality in this lab.

Note: OpenCV provides optimized functions for histogram equalization and color conversion. You are implementing your own in hopes that you will become accustomed to working with pixel values directly. For future

projects, you will be encouraged to use the built-in functionalities, and focus on putting an entire system together.

Note: Make sure you conceptually understand what you are about to implement (the above 4 steps). In a few minutes you will be diving in to a lot of details, so take a second to verify your understanding of the algorithm.

Step 1: Compute histogram of Y channel

Once you have received data in from the camera in YUV format, you will need to create a histogram of the values of the Y channel in that frame.

Note: Your histogram can be represented simply as an array. What size will the array need to be? What type?

Step 2: Compute CDF of histogram

Next, you must compute the cumulative distribution function (CDF) of your histogram. The CDF will be used in the next step to equalize the histogram.

Note: Make sure to normalize your CDF so that the range is 0 to 255.

Step 3: Apply histogram equalization

Take each Y channel value as an index into the CDF to obtain the equalized Y channel value. Read the [OpenCV histogram equalization tutorial](#) for

more information on using the CDF as a remapping function.

Note: Don't forget, you only want to equalize the left half of the image. The right half of the image must remain unmapped for comparison.

Step 4: Convert from YUV to RGB

While the pixels coming in from the camera are in YUV format, the pixels going out to the tablet's display are in RGB format. You will need to convert your half-equalized YUV image into RGB format, and store the image in the matrix pointed to by `pRGB`.

When the application is launched on the Nexus 7, you must tap the `...` near the bottom right of the screen, and select "Hist EQ". When working correctly, the right half of the video should display the unprocessed input, and the left half of the video should display the equalized video.

Extension: Other Tone Mappings

Histogram equalization is one special case of [tone mapping](#), which simulates higher dynamic range and results in more dramatic images. This section is completely optional, but if you are interested, explore and see what sort of "Instagram"-like effects you can achieve!